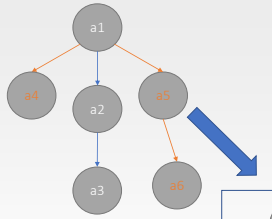
本节内容

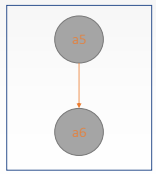
树的基本概念

树



一对一的线性结构

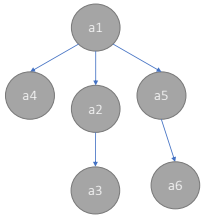
一对多的树性结构



树是递归定义的结构

树是N（N≥0）个结点的有限集合，N=0时，称为空树，这是一种特殊情况。在任意一棵非空树中应满足：
1）有且仅有一个特定的称为根的结点。
2）当N>1时，其余结点可分为m（m>0）个互不相交的有限集合T_1，T_2，…，T_m，其中每一个集合本身又是一棵树，并且称为根结点的子树。

树的相关概念

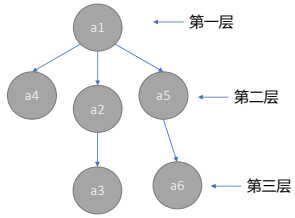


结点
- 根结点：树只有一个根结点
- 结点的度：结点拥有的子树的数量
  - 度为0：叶子结点或者终端结点
  - 度不为0：分支结点或者非终端结点
    - 分支结点中除去根结点也称为内部结点

树的度：树中所有结点的度数的最大值

结点关系
- 祖先结点：根结点到该结点的唯一路径上的任意结点
- 子孙结点
- 双亲结点：根结点到该结点的唯一路径上最接近该结点的结点
- 孩子结点
- 兄弟结点：有相同双亲结点的结点

树的相关概念



第一层

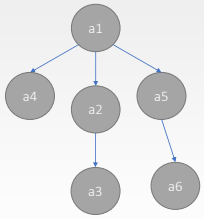
第二层

第三层

层次、高度、深度、树的高度
- 层次：根为第一层，它的孩子为第二层，以此类推
- 结点的深度：根结点开始自顶向下累加
- 结点的高度：叶结点开始自底向上累加
- 树的高度(深度)：树中结点的最大层数

树的性质

1.树中的结点数等于所有结点的度数加1。

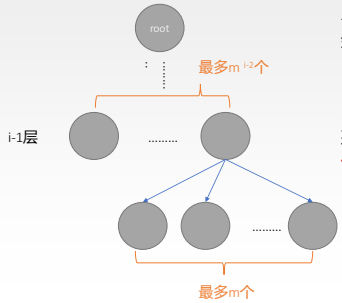


证明：不难想象，除根结点以外，每个结点有且仅有一个指向它的前驱结点。也就是说每个结点和指向它的分支一一对应。

假设树中一共有b个分支，那么除了根结点，整个树就包含有b个结点，所以整个树的结点数就是这b个结点加上根结点，设为n，则n=b+1。而分支数b也就是所有结点的度数，证毕。

树的性质

2.度为m的树中第i层上至多有m^{i-1}个结点（i≥1）。



证明：（数学归纳法）

首先考虑i=1的情况：第一层只有根结点，即一个结点，i=1带入式子满足。

假设第i-1层满足这个性质，第i-1层最多有m^{i-2}个结点。

又因为树的度为m,所以对于第i-1层的每个结点，最多有m个孩子结点。所以第i层的结点数最多是i-1层的m倍，所以第i层上最多有m^{i-1}个结点。

树的性质

3.高度为h的m叉树至多有$(m^h-1)/(m-1)$个结点

证明：（利用性质2：度为m的树中第i层上至多有m^{i-1}个结点）

所以高度为h的m叉树至多有

$N=m^0+m^1+m^2+m^3+...m^{h-2}+m^{h-1}$

$=m^0(1-m^h)/(1-m)$ (等比数列求和公式)

$=(m^h-1)/(m-1)$

4.具有n个结点的m叉树的最小高度为$\lceil \log_m(n(m-1)+1) \rceil$

最小高度时，这个树是一个满m叉树。

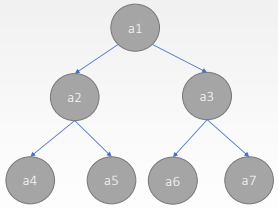
由性质3，知道满m叉树的结点数为$(m^h-1)/(m-1)$。

于是$(m^h-1)/(m-1)=n$ 解这个方程，可以得到

$h=\log_m(n(m-1)+1)$。

因为这个是高度，是整数，所以要向上取整。

满二叉树



本节内容

树的存储结构

顺序存储结构

双亲表示法：用一组连续的存储空间存储树的结点，同时在每个结点中，用一个变量存储该结点的双亲结点在数组中的位置。

```
typedef char ElemType;
typedef struct TNode{
        ElemType  data;  //结点数据
        int parent;        //该结点双亲在数组中的下标
}TNode;                    //结点数据类型
```

data	parent

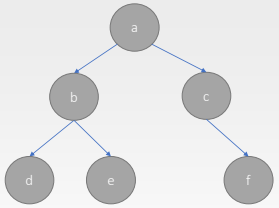
结点类型

```
#define Maxsize 100
typedef struct {
        TNode nodes[MaxSize];  //结点数组
        int n;                 //结点数量
}Tree;                         //树的双亲表示结构
```

data	parent
data	parent
.....	.....

树

顺序存储结构



下标	data	parent
0	a	-1
1	b	0
2	c	0
3	d	1
4	e	1
5	f	2

双亲表示法可以根据parent值找到该结点的双亲结点，时间复杂度为O(1)。

可是如果想找某个结点的孩子结点？

链式存储结构（1）

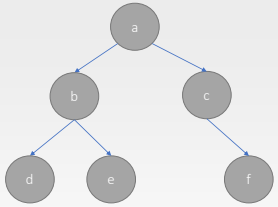
孩子表示法：把每个结点的孩子结点排列起来存储成一个单链表。所以n个结点就有n个链表；
如果是叶子结点，那这个结点的孩子单链表就是空的；
然后n个单链表的的头指针又存储在一个顺序表（数组）中。

需要设计两种结点结构类型：一是孩子链表的孩子结点，二是每个孩子链表的表头结点（存在数组中）。

```
typedef char ElemType;
typedef struct CNode{
        int child;              //该孩子在表头数组的下标
        struct CNode *next;     //指向该结点的下一个孩子结点
}CNode，*Child;                 //孩子结点数据类型

typedef struct {
        Elemtype data;          //结点数据域
        Child firstchild;       //指向该结点的第一个孩子结点
}TNode;                         //孩子结点数据类型
```


链式存储结构（1）



下标	data	parent	firstchild
0	a	-1	1
1	b	0	3
2	c	0	5
3	d	1	^
4	e	1	^
5	f	2	^

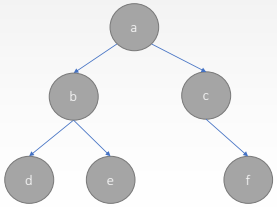
1		2	^
3		4	^
5	^		

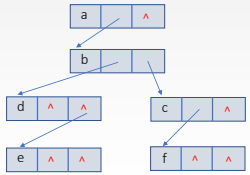
```
#define Maxsize 100
typedef struct {
        TNode nodes[Maxsize];   //结点数据域
        int n;                  //树中结点个数
}Tree;                          //树的孩子表示结构
```

链式存储结构（2）

孩子兄弟表示法：顾名思义就是要存储孩子和孩子结点的兄弟，具体来说，就是设置两个指针，分别指向该结点的第一个孩子结点和这个孩子结点的右兄弟结点。

```
typedef char ElemType;
typedef struct CSNode{
        ElemType data;                          //该结点的数据域
        struct CSNode *firstchild,*rightsib;      //指向该结点的第一个孩子结点和该结点的右兄弟结点
}CSNode；                                       //孩子兄弟结点数据类型
```





本节内容

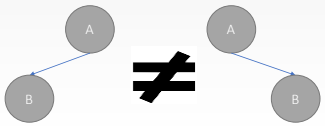
二叉树

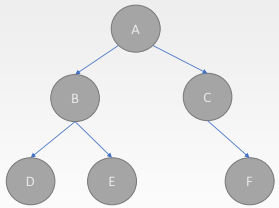
二叉树

二叉树是n（n≥0）个结点的有限集合：

① 或者为空二叉树，即n=0。
② 或者由一个根结点和两个互不相交的被称为根的左子树和右子树组成。左子树和右子树又分别是一棵二叉树。

1.每个结点最多有两棵子树。
2.左右子树有顺序





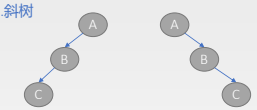
二叉树

二叉树的五种基本形态：

1.空树
2.只有一个根结点
3.根结点只有左子树
4.根结点只有右子树
5.根结点既有左子树又有右子树

特殊二叉树

1.斜树

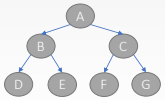


左斜树　　右斜树

每一层都只有一个结点，结点数与树的深度相同

2.满二叉树:
1)分支结点都存在左子树和右子树
2)叶子都在同一层

特点：
非叶子结点的度一定是2
相同深度二叉树中满二叉树的结点个数最多，叶子树最多

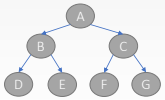


特殊二叉树

3.完全二叉树

2.满二叉树:
1)分支结点都存在左子树和右子树
2)叶子都在同一层

特点：
非叶子结点的度一定是2
相同深度二叉树中满二叉树的结点个数最多，叶子树最多



二叉树的性质

1.非空二叉树上叶子结点数等于度为2的结点数加1

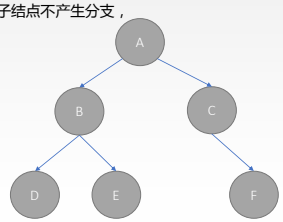
证明：设度为0,1,2的结点数分别是N_0,N_1,N_2,总结点数$N=N_0+N_1+N_2$.....(1)

考虑树中的分支数，记为B，除根结点以外每个结点都有一个分支指向该结点。所以分支数等于总结点数-1。即$B=N-1$.....(2)

度为2的结点产生两个分支，度为1的结点产生一个分支，叶子结点不产生分支，所以$B=2N_2+N_1$.....(3)

由(1)(2)(3)式，有$N_0+N_1+N_2-1=2N_2+N_1$
即 $N_0=N_2+1$

非常重要的性质！！
揭示了度为2的结点和叶子结点的数量关系。

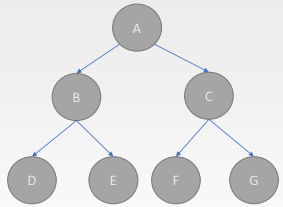


二叉树的性质

2.非空二叉树上第K层上至多有2^{k-1}个结点（K≥1）

最多的情况就是满二叉树

第一层：1个$=2^0=2^{1-1}$
第二层：2个$=2^1=2^{2-1}$
第三层：4个$=2^2=2^{3-1}$
.....
第k层：2^{k-1}



3.高度为H的二叉树至多有2^H-1个结点（H≥1）

$2^{1-1}+2^{2-1}+2^{3-1}+.....+2^{h-1}=2^h-1$(等比数列求和公式)

注意区分性质2和3。一个减1是在指数位置，一个是多项式位置。

二叉树的性质

4.具有N个（N>0）结点的完全二叉树的高度为$\lceil \log_2(N+1) \rceil$或$\lfloor \log_2 N \rfloor +1$。

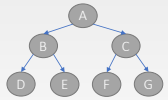
先考虑满二叉树：深度为k的满二叉树的结点数$n=2^k-1$
所以满二叉树的深度为$k=\log_2(n+1)$

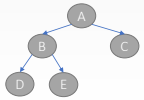
再来考虑完全二叉树，完全二叉树的结点数n一定不超过同样深度的满二叉树2^k-1，但一定多于少一层的满二叉树的结点数$2^{k-1}-1$，即满足$2^{k-1}-1<n\leq 2^k-1$。
由于n是整数$n\leq 2^{k-1}$意味着$n<2^k$，$n>2^{k-1}-1$意味着$n\geq 2^{k-1}$,于是可以改写成

$$2^{k-1}\leq n<2^k$$

两边取对数，就有$k-1\leq \log_2 n < k$
也就是 $k\leq \log_2(n+1)$且$k>\log_2 n$

由于k是整数,所以 $k=\lfloor \log_2 N \rfloor +1$（N比如为15）





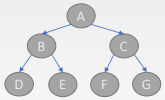
本节内容

二叉树的存储结构

顺序存储

二叉树的顺序存储结构就是用一组地址连续的存储单元依次自上而下、自左至右存储完全二叉树上的结点元素。

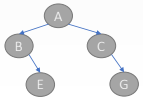
完全二叉树



下标：	1	2	3	4	5	6	7
	A	B	C	D	E	F	G

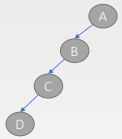
对于一般的非完全二叉树，如何来用连续存储单元存储呢？

一般二叉树



下标：	1	2	3	4	5	6	7
	A	B	C	^	E	^	G

顺序存储



$2^4-1=15$

下标：	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	A	B	^	C	^	^	^	D	^	^	^	^	^	^	^

开辟大小为15的数组却只存4个数据，为了保存二叉树的结点逻辑关系"牺牲"过大。

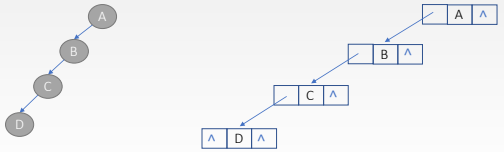
既然顺序存储"浪费"比较大，按照经验，我们会考虑链式存储结构。

链式存储

二叉树每个结点最多两个孩子，所以设计二叉树的结点结构时考虑两个指针指向该结点的两个孩子。

```
typedef struct BiTNode{
        ElemType data;      //数据域
        struct BiTNode *lchild,*rchild;  //指向该结点的左、右孩子指针
}BiTNode,*BiTree;    //二叉树结点结构
```

lchild	data	rchild



这种结构的链表由于有两个指针，所以叫二叉链表。

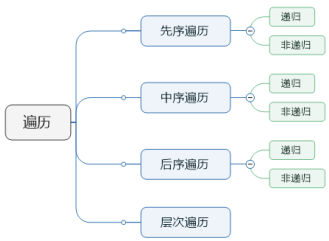
另外还可以增加指针指向该结点的双亲结点，那这时三个指针的链表就叫三叉链表。

本节内容

二叉树的遍历（递归）

二叉树的遍历

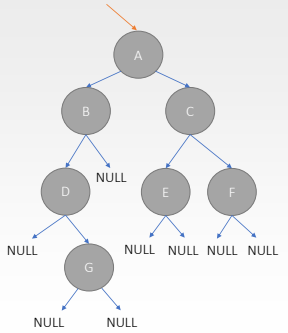
二叉树的遍历是指按某种次序依次访问树中的每个结点，使得每个结点均被访问一次，而且仅被访问一次。



递归先序遍历

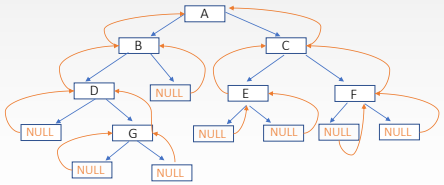
1.先序遍历的操作过程为：
如果二叉树为空，什么也不做。否则：
1）访问根结点；
2）先序遍历左子树；
3）先序遍历右子树。

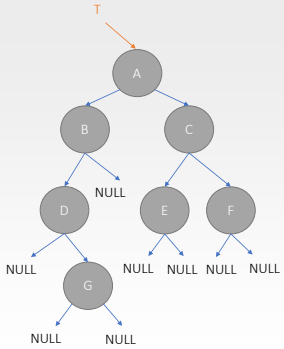
遍历序列：A B D G C E F



递归先序遍历

```
void PreOrder(BiTree T){
    if(T!=NULL){
        printf("%c",T->data)          //访问根结点
        PreOrder(T->lchild);          //递归遍历左子树
        PreOrder(T->rchild);          //递归遍历右子树
    }
}
```



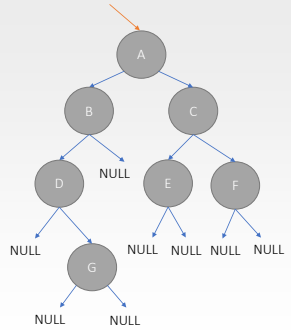


A B D G C E F

递归中序遍历

2.中序遍历的操作过程为：
如果二叉树为空，什么也不做。否则：
1）中序遍历左子树；
2）访问根结点；
3）中序遍历右子树。

```
void InOrder(BiTree T){
    if(T!=NULL){
        InOrder(T->lchild);          //递归遍历左子树
        printf("%c",T->data)；  //访问根结点
        InOrder(T->rchild);     //递归遍历右子树
    }
}
```

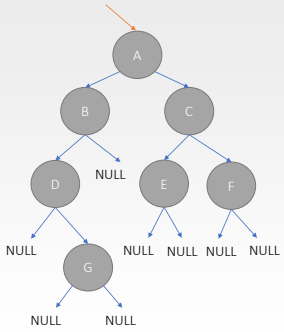


D G B A E C F

递归后序遍历

3.后序遍历的操作过程为：
如果二叉树为空，什么也不做。否则：
1）后序遍历左子树；
2）后序遍历右子树；
3）访问根结点。

```
void PostOrder(BiTree T){
    if(T!=NULL){
        PostOrder(T->lchild);   //递归遍历左子树
        PostOrder(T->rchild);   //递归遍历右子树
        printf("%c",T->data)；  //访问根结点
    }
}
```



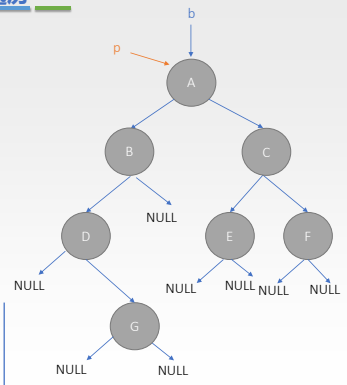
G D B E F C A

本节内容

二叉树的遍历（非递归）

非递归先序遍历

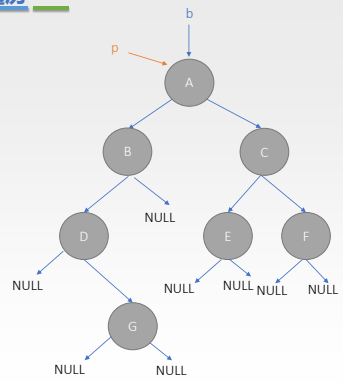
```
Void PreOrderTraverse(BiTree b){
    InitStack(S);
    BiTree p=b; //工作指针p
    while(p || !IsEmpty(S)){
        while(p){
            printf("%c",p->data); //先序先遍历结点
            Push(S,p);            //进栈保存
            p=p->lchild;
        }
        if(!IsEmpty(S)){
            p=Pop(S);
            p=p->rchild;
        }
    }
}
```



A B D G C E F

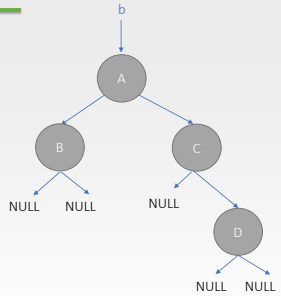
非递归中序遍历

```
Void PreOrderTraverse(BiTree b){
    InitStack(S);
    BiTree p=b; //工作指针p
    while(p || !IsEmpty(S)){
        while(p){    //中序先将结点进栈保存
            Push(S,p);
            p=p->lchild;
        }
        //遍历到左下角尽头再出栈访问
        p=Pop(S);
        printf("%c",p->data);
        p=p->rchild; //遍历右孩子
    }
}
```



非递归后序遍历

```
Void PostOrderTraverse(BiTree b){
    InitStack(S);
    BiTree p=b，r=NULL; //工作指针p 辅助指针r
    while(p || !IsEmpty(S)){
        //1.从根结点到最左下角的左子树都入栈
        if(p){
            Push(S,p);
            p=p->lchild;
        }
        //2.返回栈顶的两种情况
        else{
            GetTop(S,p); //取栈顶 注意不是出栈！
            //①右子树还未访问，而且右子树不空，第一次栈顶
            if(p->rchild &&p->rchild !=r) p=p->rchild;
            //②右子树已经访问或为空，接下来出栈访问结点
            else{
                Pop(S,p);
                printf("%c",p->data);
                r=p;       //指向访问过的右子树根结点
                p=NULL; //使p为空从而继续访问栈顶
            }
        }
}
```



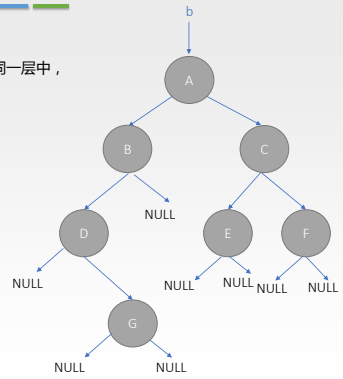
层序遍历

层序遍历：若树为空，则什么都不做直接返回。
否则从树的第一层开始访问，从上而下逐层遍历，在同一层中，按从左到右的顺序对结点逐个访问。

A B C D E F G

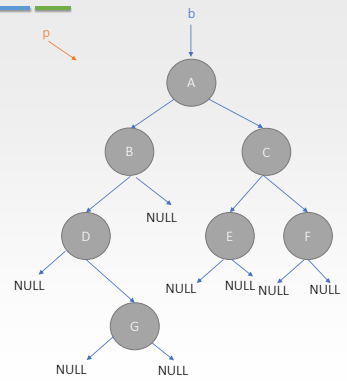
访问完一个结点之后，把它的孩子存起来，而且先访问的结点，它的孩子也是先访问。

出队 → 访问 → 左右孩子入队



层序遍历

```
void LevelOrder(BiTree b){
        InitQueue(Q);
        BiTree p;
        EnQueue(Q,b);              //根结点进队
        while(!IsEmpty(Q)){        //队列不空循环
        DeQueue(Q，p)              //队头元素出队
                printf("%c",p->data);
                if(p->lchild!=NULL)
                        EnQueue(Q,p->lchild);
                if(p->rchild!=NULL)
                        EnQueue(Q,p->rchild);
        }
}
```

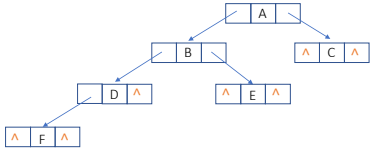


A B C D E F G

本节内容

线索二叉树

线索二叉树



二叉链表表示的二叉树存在大量空指针

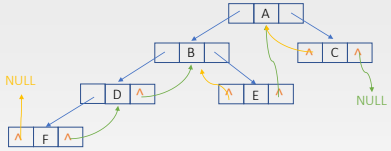
N个结点的二叉链表，每个结点都有指向左右孩子的结点指针，所以一共有2N个指针，而N个结点的二叉树一共有N-1条分支，也就是说存在2N-(N-1)=N+1个空指针。比如左图二叉树中有6个结点，那么就有7个空指针。

中序遍历序列：FDBEAC　　E的前驱是B 后继是A

非线性结构 → 遍历 → 线性结构

大量的空余指针能否利用起来？

线索二叉树



FDBEAC

后继

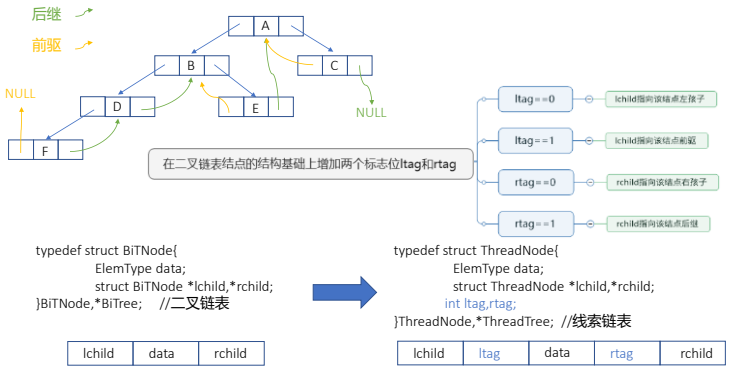
前驱

指向前驱和后继的指针称为线索，加上线索的二叉链表就称为线索链表，相应的二叉树就称为线索二叉树

对二叉树以某种次序遍历使其变为线索二叉树的过程就叫做线索化

如何区分指针是指向左孩子还是前驱，或者指向右孩子还是后继？

线索二叉树




```
typedef struct BiTNode{
        ElemType data;
        struct BiTNode *lchild,*rchild;
}BiTNode,*BiTree;    //二叉链表
```

lchild	data	rchild

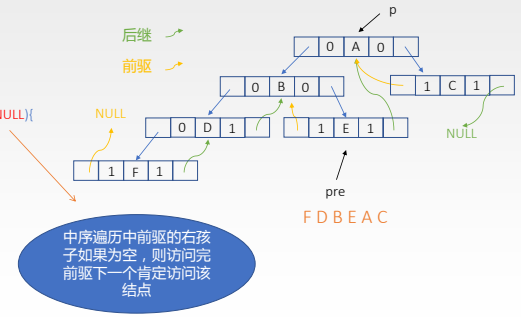
```
typedef struct ThreadNode{
        ElemType data;
        struct ThreadNode *lchild,*rchild;
      int ltag,rtag;
}ThreadNode,*ThreadTree;  //线索链表
```

lchild	ltag	data	rtag	rchild

构造线索二叉树

```
void InThread(ThreadTree &p,ThreadTree &pre){
//中序遍历对二叉树线索化的递归算法
  if(p){
        InThread(p->lchild,pre);
        if(p->lchild==NULL){
            p->lchild=pre;
            p->ltag=1;
        }
        if(pre!=NULL&&pre->rchild==NULL){
            pre->rchild=p;
            pre->rtag=1;
        }
        pre=p;
        InThread(p->rchild,pre);
    }
}
```

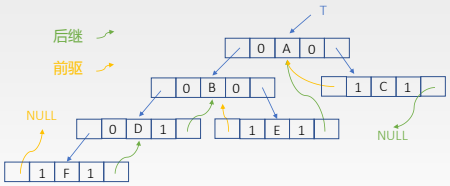
pre指向中序遍历时上一个刚刚访问过的结点　初值为NULL



中序遍历中前驱的右孩子如果为空，则访问完前驱下一个肯定访问该结点

遍历线索二叉树

```
void InOrderTraverse(ThreadTree  T){
      ThreadTree p=T；
      while(p){
          while(p->ltag==0)p=p->lchild;
            printf("%c",p->data);
          while(p->rtag==1&&p->rchild){
                p=p->rchild;
                printf("%c",p->data);
           }
           p=p->rchild;
      }
}
```



F D B E A C

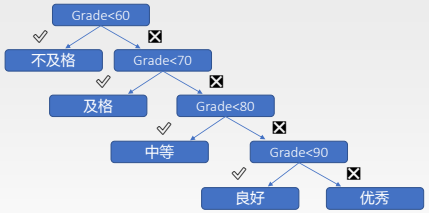
本节内容

哈夫曼树和哈夫曼编码

哈夫曼树

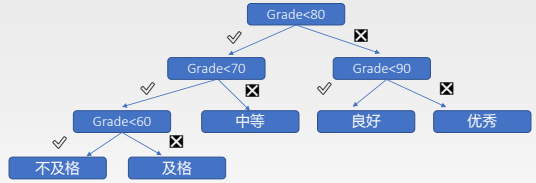
成绩评定

```
if(grade<60) return “不及格”;
else if(grade<70) return “及格”;
else if(grade<80) return “中等”;
else if(grade<90) return “良好”;
else return “优秀”;
```



大部分学生成绩都是处于70到90之间 → 大部分人都需要至少三次判断

哈夫曼树



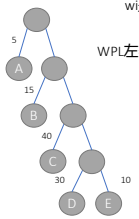
大部分学生成绩都是处于70到90之间 → 大部分人只需要两次判断

如何能设计这种效率最高的树结构？

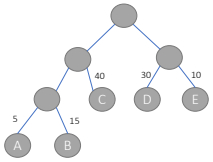
哈夫曼树

树中所有叶结点的带权路径长度之和称为该树的带权路径长度 $WPL=\sum_{i=1}^{n} w_i \times l_i$

w_i是第i个叶结点所带的权值；l_i是该叶结点到根结点的路径长度



WPL左=5*1+15*2+40*3+30*4+10*4=315



WPL右=5*3+15*3+40*2+30*2+10*2=220

哈夫曼树

哈夫曼树：含有N个带权叶子结点的二叉树中，其中带权路径长度（WPL）最小的二叉树，也称为最优二叉树。

如何设计哈夫曼树？

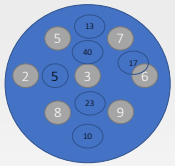
算法的描述如下：
1）将这N个结点分别作为N棵仅含一个结点的二叉树，构成森林F。
2）构造一个新结点，并从F中选取两棵根结点权值最小的树作为新结点的左、右子树，并且将新结点的权值置为左、右子树上根结点的权值之和。
3）从F中删除刚才选出的两棵树，同时将新得到的树加入F中。
4）重复步骤2）和3），直至F中只剩下一棵树为止。

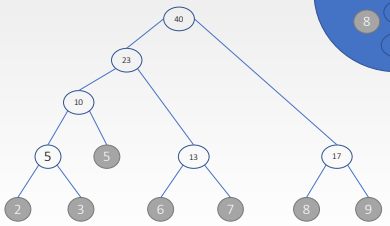
森林是m（m≥0）棵互不相交的树的集合.

设给定权集w={5，7，2，3，6，8，9}，试构造关于w的一棵哈夫曼树，并求其加权路径长度WPL。

哈夫曼树

设给定权集w={5，7，2，3，6，8，9}，
试构造关于w的一棵哈夫曼树，
并求其加权路径长度WPL。





1）每个初始结点最终都成为叶结点，并且权值越小的结点到根结点的路径长度越大。
2）构造过程中共新建了N-1个结点（双分支结点），因此哈夫曼树中结点总数为2N-1。
3）每次构造都选择2棵树作为新结点的孩子，因此哈夫曼树中不存在度为1的结点。

WPL=2*4+3*4+5*3+6*3+7*3+8*2+9*2=108

哈夫曼编码

哈夫曼编码最早用于解决远距离电报通信的数据传输最优化问题。

比如ABCDE这一段文字。通过网络传输要用到二进制的数字0和1。来保存A,B,C,D,E 五个字母就可以了。

字母	A	B	C	D	E
二进制字符	000	001	010	011	100

ABCDE　　000 001 010 011 100

如果传输的信息很长，那译码也会很长

例如：AABBCCDDEE.....　　000 000 001 001 010 010 011 011 100 100.....

不同的字符和字母出现的频率实际上是不一样的。

'A'，'E'，'I'，'O'，'U'

"有"，"是"，"在"

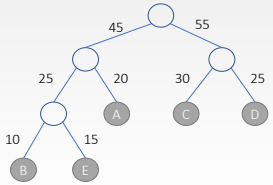
哈夫曼编码

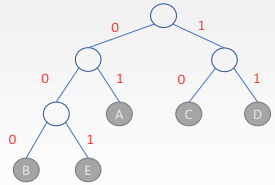
字母出现的频率：A：20% B：10% C：30% D：25% E：15%

字母	A	B	C	D	E
频率	20	10	30	25	15

构造哈夫曼树：

将权值左分支改为0，右分支改为1





从根结点走到每个叶子节点

字母	A	B	C	D	E
编码	01	000	10	11	001

哈夫曼编码

等长编码的译码　　AABBCCDDEE.....　　000 000 010 010 011 011 100 100 101 101.....

前缀编码:如果没有一个编码是另一个编码的前缀。

0、101、100、

非前缀编码：A：10 B：101 C：110

10110.....

①10 110　　AC
②101 10　　BA

ABCDE

哈夫曼编码

字母	A	B	C	D	E
编码	01	000	10	11	001

010001011001 共12位

节约约20%的成本

等长编码

字母	A	B	C	D	E
编码	000	010	011	100	101

000010011100101 共15位